L15 - Abstract Classes and Interfaces

March 5, 2020

1 More Abstraction: Abstract classes and Interfaces

1.1 Simulation

Many programs are used for simulation in science, engineering, medicine and economics. Generally these are only partial simulations; they include simplifications because the underlying system is only empirically understood. The greater the detail, generally the more processing time and the more programming effort is needed.

Benefits

- support useful predictions
- allow experiments that would be precluded for ethical or economic reasons

1.2 `foxes-rabbits`

A predator-prey model for foxes and rabbits, simulating if a highway is built. Run this in an IDE; it's like 17 files and a nightmare to try and paste in here.

1.2.1 `Rabbit`

Simple model of a prey species.

State

- age: how old in this `Rabbit`?
- `alive`: is this `Rabbit` still kickin? False means a `Fox` ate it.
- `location`: where is the `Rabbit`?
- `field`: the field this `Rabbit` lives on

`Rabbit` state is managed by the `run` method. This method might cause the `Rabbit` to breed or die of old age.

1.2.2 `Fox`

Simple model of a predator

State

- everything `Rabbit` has and
- `foodLevel`: how full of `Rabbit` is this `Fox`? Increased by eating `Rabbits`.

`Fox` state is mananged by the `hunt` method.

1.2.3 `Simulator`

Manages the overall simulation task. Holds collections of `Foxs` and `Rabbits`.

State

- constructor setup
- `populate`
  - each animal is given a random starting age
- `update`

1.2.4 `Field`

Represents a 2d grid

1.2.5 `Location`

Represents a 2d location on the `Field` with a row and column value.

1.2.6 `FieldStats`, `Counter`

Keep track of the statistics.

1.2.7 `Randomizer`

Generate random seeds for the simulation - randomizing starting conditions and

1.2.8 Room for improvement

- a lot of commonality between `Fox` and `Rabbit`
- the `Simulator` needs to know a lot about `Foxes` and `Rabbits`.

1.3 `Animal` superclass

We could place common field like `age` and `alive` in `Animal`; then rename `run` and `hunt` for information hiding. `Simulator` can now be decoupled from the objects it acts on.

1.3.1 `act()`

Static type checking requires there be an `act()` method in `Animal`. However, the desired outcomes are very different in `Fox` and `Rabbit`: there is no obvious shared implementation. Instead we can declare `act()` as an *abstract* method: a method *with no body*.

1.4 Abstract Classes and Methods

- abstract methods have `abstract` in the signature
- abstract methods have no body
- abstract methods make *the entire class abstract*
- **abstract classes cannot in instantiated** - you cannot create an object from them
- concrete (that is, not abstract) subclasses complete the implementation

1.5 Implementing `Animal`

[ ]:

```java
public abstract class Animal
{
    // fields omitted

    abstract public void act(List<Animal> newAnimals); // SEMICOLON HERE!!!
}
```

1.6 Extending the simulation

What if you also have a `Hunter`? That's not an animal, it probably acts differently. Instead, we can create another abstract superclass `Actor` that includes things common to being on the field.

1.6.1 Multiple inheritance

Say you also wanted to have your simulation support `Ants` that behave like `Animals` but cannot be drawn on the grid. You could have a class `Drawable` that deals with grid operations. What do you do with `Rabbit`?

This cannot be done in Java - a single class *cannot* inherit from two classes simultaneously (called "multiple inheritance"). However, Java permits multiple inheritance for *interfaces*.

1.7 Interfaces

An interface is essentially a chunk of method prototypes that constitute a contract about the function of an object with the outside world and end users which is enforced by the compiler at build time.

A Java interface:

- uses `interface` rather than `class` at declaration
- does not contain a constructor
- contains no instance fields
- only fields that are constant class fields with `public` visibility are allowed
- abstract methods do not need to include `abstract` in their header

1.7.1 Default methods

Methods marked `default` in an interface have a body, which will be inherited by all inheriting classes.

Classes that inherit from two different interfaces which have default methods with the same signature must override that default method.

1.7.2 Interface as specifications

Interfaces separate functionality from implementation strongly; the client-side interaction is entirely separate from the implementation and allow clients to choose different implementations.

`List`, `LinkedList` and `ArrayList` are examples of this.

[1]:

```
public interface Actor
{
    void act(List<Actor> newActors);
}

public class Fox extends Animal implements Drawable
{
    // class body
}
```

1.8 The `Class` class

A `Class` object is returned by `getClass()` in `Object`. The `.class` suffix provides a `Class` object (example: `Fox.class`).

[ ]:

```
```